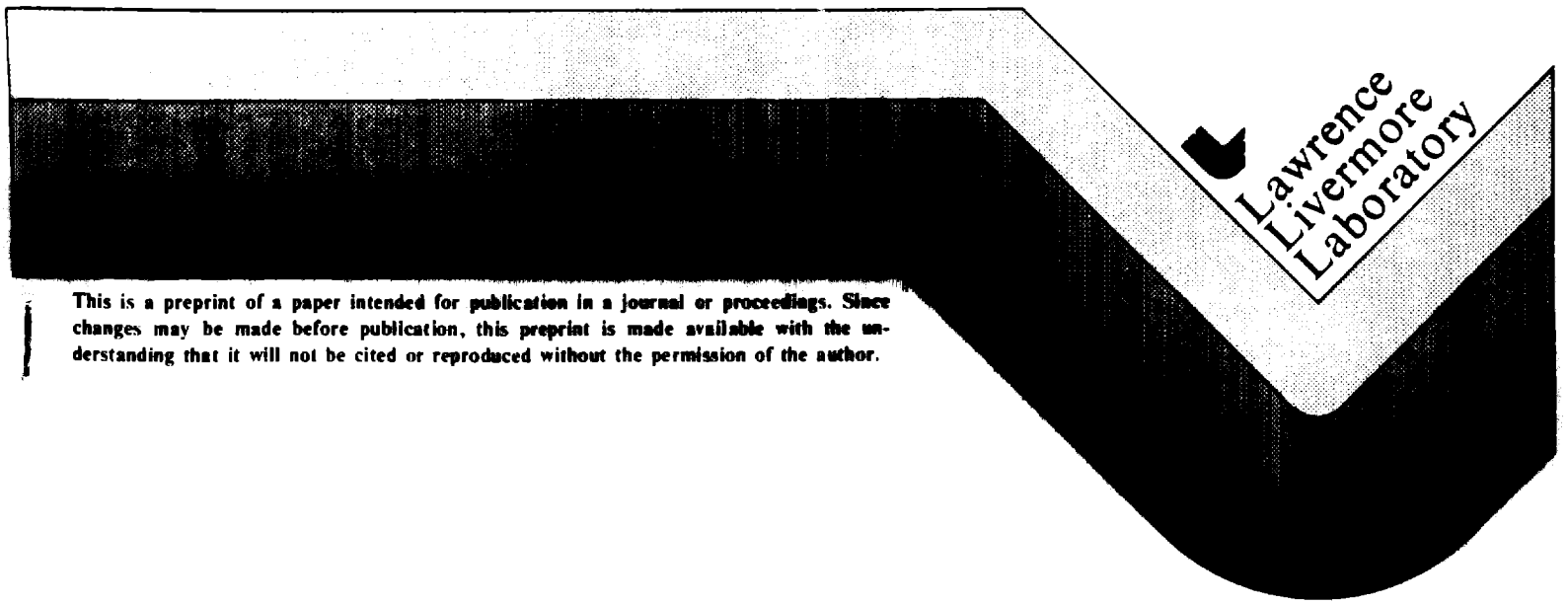
UCRL- 87494
PREPRINT

The S-1 Multiprocessor System

J. M. Broughton
P. M. Farmwald
T. M. McWilliams

This Paper Was Prepared For Submittal To
SPIE Technical Symposium East '82
Arlington, Virginia
May 3-7, 1982

April 2, 1982



DISCLAIMER

341 45

The S-1 Multiprocessor System

J. M. Broughton, P. M. Farmwald, T. M. McWilliams

Lawrence Livermore National Laboratory
P.O. Box 5503, Livermore, California 94550

Abstract

This paper describes the S-1 multiprocessor system. It is composed of 16 supercomputer class uniprocessors with local caches, an extremely large, medium latency shared memory, and a low latency synchronization bus for passing short messages. The system is applicable to a wide variety of applications, including large-scale physical simulation, real-time command and control, and program development in a time-sharing environment. The hardware organization, its implications, and software supporting the efficient utilization of the multiprocessor are discussed.

Introduction

The S-1 Project [1] is engaged in the development of advanced digital processing technology for potential application in the military and scientific communities. Current work being sponsored by the U.S. Navy and the Department of Energy involves the design and development of extremely high performance, general purpose computers (S-1) and multiprocessor interconnection technologies.

The reasons for development of multiprocessors have been widely discussed; chief among them are reliability, economy and scale. We place heavy — though not exclusive — emphasis on the issue of scale.

Today, there are a number of important problems for which manual solution is infeasible, yet cannot be handled by existing computers because they have insufficient computing power [2]. As an example, the ability to provide an accurate two week weather forecast would have extraordinary economic leverage. It would allow farmers to select optimal times for planting and harvesting, and provide substantial warning of natural disasters to minimize loss of life and property. The latest computational methods for weather prediction are believed to be adequate for the task; unfortunately, they overwhelm the computing and storage capacity that is available today. Develoment of new oil and mineral resources is of vital national importance. Much of the exploration being conducted involves seismic data processing, and employs vast computer resources. Effective utilization of the new semiconductor technology, specifically very large scale integration (VLSI), is limited by our ability to design and debug circuits involving hundreds of thousands of transistors. Computer-aided design techniques, such as the Project's SCALD system [3], have been demonstrated to greatly reduce development time of new digital systems; however, their use is effectively limited to designs of moderate size because of capacity limitations. Similar limitations are seen in a variety of military applications.

Given a particular logic technology, there is a limit to performance that can be obtained regardless of the complexity or cleverness of the processor design. Today's fastest processors using commercially available components have a peak performance in the 10-40 MIPS (million instruction per second) regime for scalar operations and 100-400 MFLOPS (million floating point operations per second) for vector operations. A multiprocessor, however, can exceed the inherent limitations on a single processor by performing computations in parallel.

Multiprocessor systems which have been demonstrated to date fall into roughly two categories. The first includes systems that have a large number of small scale processors (minicomputers or microcomputers). Examples include most of the early research multiprocessors such as CM*. Aggregate system performance is limited because of the limited performance of the processing elements, and the limited number of processing elements connected together. The second category encompasses systems that have a small number of medium scale processors (small mainframes). This approach has been taken in several commercial offerings that provide cost effective performance enhancement for batch or time-sharing applications through dual-processor configurations. Aggregate system performance is not an issue in these systems as they are used to run more jobs rather than a single job faster.

The S-1 Project is taking the unique approach of assembling a multiprocessor consisting of up to sixteen uniprocessors, each of which have a performance comparable to that of the fastest supercomputers. This paper will address four topics: design of the uniprocessors, the multiprocessor architecture, operating system support, and the tools for partitioning

single problems for a multiprocessor.

Uniprocessors

For use in the multiprocessor, we are developing a family of processors having similar architectures, but differing implementation technology. Each successive family member is intended to make maximally effective use of the then available logic families. Such a succession of processors is required in order to maintain the multiprocessor's edge. Advances in semiconductors are occurring at such a rapid rate that a multiprocessor tied to one particular technology would soon be made obsolete by single processors having an order of magnitude greater speed.

The first generation of the S-1 family of processors is the Mark I, which has been operational since 1978. Implemented in ECL-10K medium scale integrated circuits (MSI), it is roughly equivalent in processing power to one-third of a CDC 7600. The second generation, the Mark IIA, is currently undergoing initial checkout. Through use of extensive hardware support for vector and floating point computations, and faster logic (ECL-100K MSI), it is expected to achieve performance comparable to existing supercomputers such as the Cray-1. Future generations are planned that will follow the leading edge of implementation technologies to obtain ever increasing performance and ever decreasing cost, power and space requirements. The S-1 Mark V, targeted for development in 1985, is intended to be a "supercomputer on a wafer" with performance 2-3 times that of the Mark IIA.

Unlike traditional supercomputers which sacrifice functionality for performance, the architecture of the S-1 uniprocessors has been designed to be easy and efficient to use for a wide variety of applications. In this, it closely resembles the highly popular mini-mainframes which stress flexibility over performance.

The architecture was designed with a number of goals in mind. First, it must be suitable for high performance implementation; second, it must be simple for a high level language (e.g. Ada) compiler to make effective use of instruction set; and third, it must provide a comprehensive set of data types and operations so that the programmer can select the arithmetic precision appropriate to a problem.

In addition, to the usual general purpose features, the S-1 architecture has incorporated a number of special purpose operations to provide especially high performance for its anticipated applications. Many scientific codes make heavy use of elementary functions such as sine, cosine, exponentials, and logarithms. The architecture provides these functions as single instructions, and the Mark IIA has special hardware to permit the instructions to execute at about the same speed as a simple multiply. An extensive vector instruction set is provided to enhance performance on problems that manipulate large arrays of data. Special vector instructions are provided for signal processing applications. Examples include FFT's and filtering operations. Matrix operations are also supported, including matrix multiply and generalized transpose. Because the S-1 implementations are uniformly cached-based, all vector instructions execute with a one element step size to avoid inefficient use of the cache. In cases where the problem requires non-unity step sizes, the transpose instruction can be used to extract the relevant elements into a unity step size temporary vector.

The S-1 architecture provides the user with a large, segmented virtual address space spanning 2 billion 9-bit bytes of data. Memory capacity on this scale is crucial for the effective solution of large problems such as three-dimensional physical simulations. The large address space allows all the problem data to reside directly in memory in the obvious fashion, and eliminates the programming contrivances needed to explicitly manage multiple types of computer system storage (i.e., manually swapping data to and from a disk file). A virtual memory mechanism maps the virtual address space to physical memory. In the event that the user's memory requirements exceed physical capacity, it is possible for the operating system to simulate the additional memory with a slight performance penalty; this avoids the problem of a program "falling off a memory cliff". With today's rapidly decreasing costs of memory, however, it is economical to purchase sufficient memory to meet the requirements of even the largest programs.

Multiprocessor Systems

The S-1 Multiprocessor System is a MIMD (multiple instruction, multiple data) stream organization. The multiprocessor currently being built at the Lawrence Livermore National Laboratory consists of 16 Mark IIA processors, connected together with a crossbar switch as shown in Figure 1.

A crossbar is the highest possible performance interconnection network, with a direct logical connection from each processor to each memory bank. Given that high performance processing elements are being used, the cost of the crossbar switch turns out to only be a few percent of the system cost, making it the obvious choice for use.

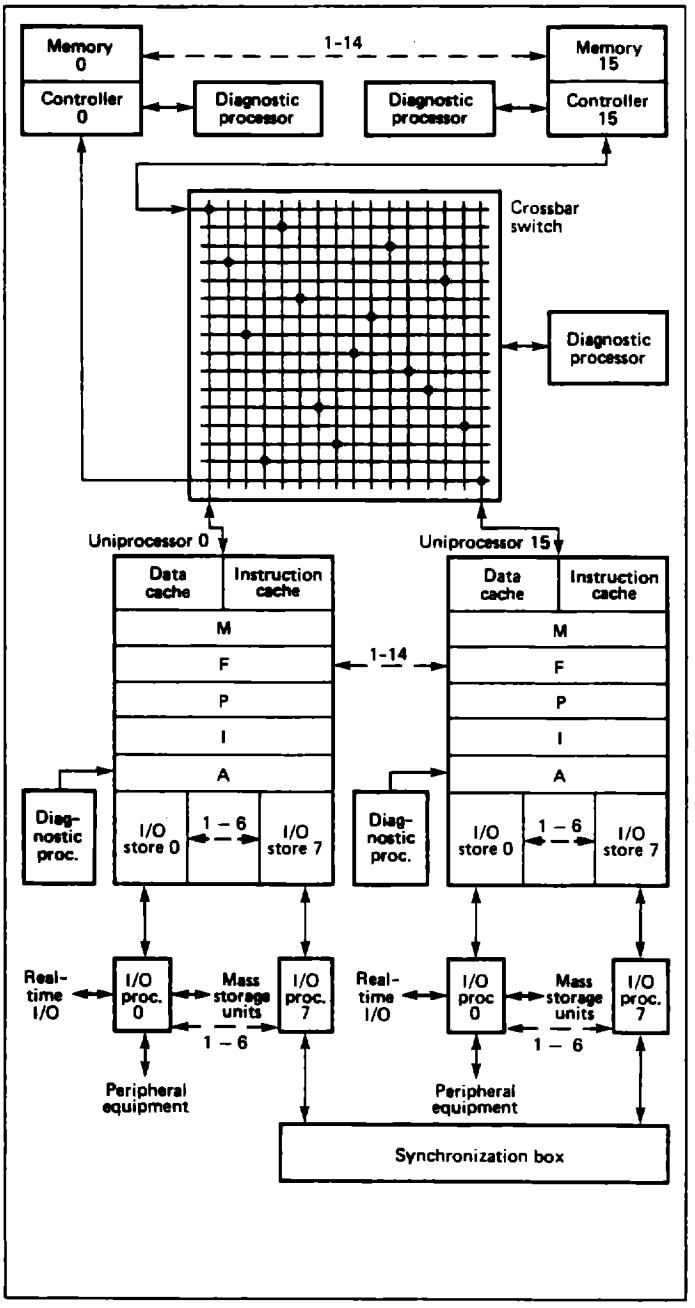

Memory 0
1-14
Memory 15
Controller 0
Diagnostic processor
Diagnostic processor
Controller 15
Crossbar switch
Diagnostic processor
Uniprocessor 0
Uniprocessor 15
Data cache
Instruction cache
M
F
P
I
A
1-14
Diag-nostic proc.
I/O store 0
1 – 6
I/O store 7
Real-time I/O
I/O proc. 0
Mass storage units
I/O proc. 7
1 – 6
Peripheral equipment
Synchronization box

Figure 1.

To the programmer, the S-1 Multiprocessor looks like 16 identical processors executing out of a very large (up to 16 billion bytes) common memory. The processors always get the latest value associated with a memory location, and instructions operate in a read-modify-write fashion. All of the complexity of moving results between different processors and between processors and memories are completely handled by the hardware in an invisible fashion.

In order to speed up effective memory access times, the processor keeps the most-recently referenced memory locations in cache memories, which are very high-speed local memories contained inside of the processors. Each processor has two cache memories, one which keeps track of the most-recently referenced 64K bytes of data, and one which keeps track of the most-recently referenced 16K bytes of instructions. When a processor wants to use a location that is contained in one of the caches, the effective access time is zero, since the read is overlapped with the execution of the previous instruction. When the processor wants to use a location that is not contained in one of the caches, the processor goes out to the main memory to see if the desired location is stored there. If so, it reads the 64 bytes around the location of interest, and stores them in its cache for future use, removing the 64 bytes that haven't been referenced for the longest period of time. If the location that is desired is contained in the cache of another processor, the requesting processor will ask the processor that has the location in its cache to remove it from its cache, and to transmit the data to the requesting processor.

The technique by which the hardware automatically keeps track of shared data in a multiprocessor with caches is called cache coherence [4]. Associated with each block of 64 bytes in main memory are an additional 17-bits that specify the current "ownership" of the block. There is one bit for each of the 16 processors in the multiprocessor which is set if the corresponding processor has a copy of the block in its cache, and the 17th bit says that somebody has a copy of the block for write access. Multiple processors are allowed to have copies of a block for read access, but only one processor is allowed to have a block for write access.

The use of shared memory to provide high speed synchronization and low latency data transmission (less than a microsecond) is difficult. For problems which require very close cooperation between the processing elements, a special set of hardware implemented queue instructions are provided. These instructions allow one processor to put computed results into a queue for another processor, which takes values and does further computation. We have found that this can substantially speed up processing in some algorithms.

Operating System Support

In order to provide a workable software base for experimentation, the S-1 Project has undertaken the development of a new operating system, called Amber, that is intended to provide a flexible interface to the multiprocessing capabilities of the system.

The basic design goal of Amber is to support a widely varying community of users — including real-time, computation intensive, and time-sharing — on one system. We see this as particularly important since it allows for extensive sharing of effort, both in the development of system software and applications software. Often several operating systems are developed for new computers, one for each major class of application. When this occurs, there is little motivation to share development effort between the different operating systems. Facilities with common functions are implemented multiple times with different interfaces for each operating system. This not only increases the total development burden, but also limits the rate at which the system matures, since a smaller user group is available to test out the system. On the other hand, when there is a single multi-function operating system; tools such as compilers, debuggers and file maintenance utilities can be readily shared between different applications. More important is the fact that libraries developed by user groups can be shared as well. There is, however, a danger in the development of a multi-function system; the system may not fulfill any of the requirements well. To avoid this problem, Amber has a modular layering of functions. The lowest levels of the system provide only atomic functions that can be implemented efficiently; higher levels of the system incorporate the more complex functions which are specific to particular applications. Commonality is therefore retained, but an application need only invoke the functions necessary to it.

The central example of this kind of layering of function exists in the scheduler. The low level scheduler provides an efficient mechanism for short-term scheduling of tasks on a single processor. The basic algorithm is a simple priority scheduling algorithm with round-robin queues. Within a single priority level, each task may run until it must wait for some external event, or an assigned run quantum expires, at which time it is moved to the end of the queue. Tasks may be assigned to different priority levels depending on their relative importance or real-time constraints. For example, in a time-sharing system normal user tasks might be given priority over batch jobs, and relinquish priority to tasks which

must respond to external interrupts in a certain length of time. The high level scheduler implements higher level, policy oriented scheduling functions, by manipulating the parameters of the low level scheduler, such as task priority or quantum size. The simplest example of such a scheduling policy is for "real-time" jobs. Here the policy is simple, select the priority that the job is to have, and assign it to a processor. More complex policies occur in batch or time-sharing systems, where it may be desirable to load-share across all the processors in the system or to guarantee a particular job a certain fraction of system resources. In contrast to the low level scheduler, which makes assignments on millisecond timescales, the policy decisions are made on second or minute timescales and can therefore be relatively expensive without unduly affecting system response.

The low level scheduler enforces a dedicated processor assignment for each task given to it, rather than scheduling each task to the next available processor. This means that processors may lie idle in the system while there are tasks ready to run. While this may seem unfortunate at first glance, there is in fact strong motivation to restrict task to single processors in the short run. First, the I/O architecture of the S-1 attaches peripherals through dual-port memories to a particular processor. A task whose purpose is to control a peripheral can only run on the processor to which the peripheral is attached; the task's processor assignment must reflect this fact. Second, the internal processor caches are very large, and as a task runs it builds up a substantial investment in data that has been locally cached. If its execution were to be moved to a different processor, the data would have to be swapped back to main memory and then swapped into the cache of the new processor. Consequently, if tasks are moved from one processor to another on a short time scale, a noticeable performance degradation results. By performing processor reassignment on a relatively long time scale, the effect is trivialized. Third, to support parallelism between tasks working on the same problem, it is necessary to insure concurrency of execution. By assigning each such task to its own processor, we can do so without use of a complex algorithm which would interfere with the simple requirements of other applications.

One of the important features of the S-1 multiprocessor is the large shared memory which permits high bandwidth communications between tasks. Access to the shared memory is provided in two ways: sharing of entire address spaces, and sharing of specific data objects.

The address space of a task encompases all data to which it has access; this includes program instructions, common blocks, and local variables. In many operating systems, each task is assigned a unique address space (usually called a core image). Each task sees its own private copy of all programs and data. Modifications made are not apparent to other tasks. In Amber it is possible for two, or more, tasks to share the same address space, i.e., identically the same physical storage. As a result, a modification to data made by one task is immediately visible to another, even if the tasks reside on different processors. Such shared data may be used as semaphores or locks to synchronize the execution of the tasks or as shared data bases to be concurrently processed by the tasks.

While there are uses for sharing entire address spaces — for instance, the semantics of Ada require tasks to execute in the same address space, there is added protection in only sharing that portion of address which is actually common. For this, Amber implements segmentation. A segment is little more than an ordinary file, except that a task can instruct that the file be mapped into its address space so that it may be directly modified. When two tasks both map the same segment, they share a single physical copy, while other portions of the address space stay private. Thus, a task is protected against inadvertent modifications to its private state. The segmentation mechanism can provide further intertask protection as well. The modes of access (read, write) that a task is permitted to a segment is the same as the normal file protection. It is then possible, for example, to set up — and enforce — a reader/writer relationship between two tasks by granting one read/write access on the segment, and the other only read access.

The inherent redundancy of the multiprocessors is often used to obtain increased reliability and availability. Amber uses a facility called dynamic reconfiguration to exploit the redundant components of the S-1 multiprocessor. At any time, Amber is capable of providing service with only a partially operating configuration, and it is possible to dynamically change the configuration without halting the system. If a memory box is to be removed, data in that box is moved either to another box or to disk and the virtual memory mapping updated to reflect is new location. If a processor is to be removed, tasks on that processor are halted and redistributed to other processors for execution. When a memory or a processor is added, it is added to the pool of system resources and is assigned as needed.

A Multiprocessor Software Tool

The construction and maintenance of large application programs for a multiprocessor system presents many problems. The details of the multiprocessor system may greatly

influence the structure of the software that yields the best performance. For instance, the algorithm that is fastest on a uniprocessor may not exploit the capabilities of a multiprocessor as well as one tailored for parallelism. In addition, the number of processors available on a time-shared (or gracefully degrading) multiprocessor may vary with time, with different algorithms being appropriate for different load factors and numbers of processors.

Thus, it is desirable to maintain a single source which works well on many configurations. However, including too many details of how to best perform a task may lead to unreadable, unmodifiable, and untransportable programs. The approach taken in the Paralyzer, a tool being developed by the S-1 Project, is to split programs into two conceptual pieces. One part describes the "how" of the computation, which includes the basic data and control flow of the algorithm. The second part is the "where" of the computation. This basically specifies what processors are to run the computations, as well as modifying the control and data flow of the first section in ways appropriate to the hardware available. The intent is that the first section is relatively machine and configuration independent, whereas the second is completely driven by the computation resources.

The current version of the Paralyzer uses Pascal as the source language, and is implemented as a source-to-source translator. Special Pascal comments are used to describe some of the "where". The special comments and the "where" description file are implemented in Maclisp (a variant of Lisp) — thus a complete programming language is available for program manipulation. A library of routines have been written in Maclisp to implement the most common kinds of transformations.

A simple example of a transformation performed by the Paralyzer involves partitioning the processing of a matrix among several processors. The directives in the special comments instruct that the matrix be divided into several equal sections, along columnar boundaries, and that each section be given to a separate processor for parallel execution. Such a transformation is feasible when the computation of a single matrix element depends only on other elements in the same column; not on elements in the same row. This restriction insures that a single processor has all the data it needs to perform its part of the computation. When other dependencies exist in data, other more complicated transformations are called for.

In addition to generating code for uniprocessor and multiprocessor systems, the Paralyzer has been used to generate code for simulation purposes. For instance, instead of generating variable references, the Paralyzer can generate calls to routines that allow the cache and memory performance of the algorithm to be determined.

Summary

The S-1 Multiprocessor System is a new step in the development of high-performance computer systems. It combines many cost effective supercomputers with the interconnection hardware and software to effectively utilize those processors on a single problem. The goal is to provide the capability to solve real problems of interest to real users.

Acknowledgments

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W-7405-ENG-48, with support from the Naval Electronics Systems Command and the Office of Naval Research.

References

1. S-1 Project Staff, "Advanced Digital Processor Technology Base Development for Navy Applications: The S-1 Project," Lawrence Livermore Laboratory, Report UCID 18038 (1978).
2. Levine, R. D., "Supercomputers," Scientific American, Vol. 246, No. 1, January 1982.
3. McWilliams, T. M., Widdoes, L. C., "The SCALD Physical Design Subsystem," Proceedings of the 15th Annual Design Automation Conference, Las Vegas, 1978 (IEEE, ACM, New York, 1978) p. 271.
4. Censier, L. M. and Feaurier, P. A., "A New Solution to Coherence Problems in Multicache Systems," IEEE Transactions on Computers, C27 (12), 1112 (1978).